# Push Model XML Parser / Node Factory

**Development:** Chris Lovett
**Program Manager:** Charles Frankston
**Last Update:** 1998.06.18
**Version:** 1.21 (IE5 Beta 1 version is considered 1.0)
**Location:** http://xmlweb/specs/xml/NodeFactory.htm
**Special thanks to:** Gary Burd, Johann Posch, and Scott Cottrille

## Beta 1

We don't recommend writing new code to use the NodeFactory interfaces exposed in Beta 1, because they'll be changing. However, if you must get started, the old spec is still at: http://xmlweb/msxml/parser/xmlparser.htm.

## New features/changes for IE5 Beta 2 milestone:

We are trying to make it simpler to create node factories in this release. Toward this end we're making a few changes:

**Imporant update: not all of these features are on the schedule for the IE5 Beta 2 milestone, but we're leaving them in the spec. Eventually, we'll want to do it all. Features that are not likely to be implemented for IE5 are greyed out. The schema processing, validation support, and entity expansion will not be in IE5. However, the flag values are reserved, and CreateNode will have a reserved parameter that we eventually intend to use the pass the schema argument down.**

1. Some customers want to take advantage of our Schema (DTD) processing and namespace support. Currently, our DTD & namespace processing are too intimately tied to our own IDOMNode factory. It is not possible for a customer NodeFactory to ask for namespace and/or schema processing – either you use our IDOMNode interface, or you do all your own namespace, schema processing, and entity expansion. This revised spec documents two parser flags XMLFLAG_PROCESSNAMESPACES and XMLFLAG_PROCESSSCHEMAS, which allow a customer node factory to request namespace and schema processing.
2. We have separated the dwType parameter on IXMLFactory::CreateNode into a dwType and a subType. This makes it easier to uniformly process (or ignore) entire classes of tags.
3. There is a new parser flag: XMLFLAG_EXPANDENTITIES. If this flag is off, the behavior is as it was for Beta 1 -- the customer NodeFactory receives CreateNode calls on ENTITYREF objects (i.e. for each "&foo;"). If this flag is set, the parser expands all entity references for the customer node factory. This makes it easier to write the common case where the customer NodeFactory application need not be aware of what content came from entities as opposed to in-line text.
4. The IXMLParser interface is broken out into INodeSource, and IXMLParser. The idea is to allow an application to act as a NodeSource and drive an INodeFactory without having

to implement a full IXMLParser interface.

# Introduction -- should you use the Node Factory?

This document defines a low level interface for manipulating XML without having to build an in-memory tree representation of the entire XML document. This interface is ideal for customers who want to scan a chunk of XML for a certain tag. For example, the CDF Viewer needs to scan every CDF file in a given folder for a LOGO tag that has an ICON attribute so it can display the appropriate icon. Clearly it needs to do this in the most efficient manner possible.

It is called the **Push Model** parser because the client has to keep pushing it to get the job done. This way the client can parse the XML asynchronously without using multiple threads or fibers.

These are low level interfaces that are designed for C programmers only and will not be scriptable. First there is an **IXMLParser** interface which is used to parse XML either from an IStream or directly from an in-memory buffer. Then there is a **Node Factory** callback interface that the parser calls for each XML tag or attribute it finds in the source. The client implements their own Node Factory.

**Alternatives**

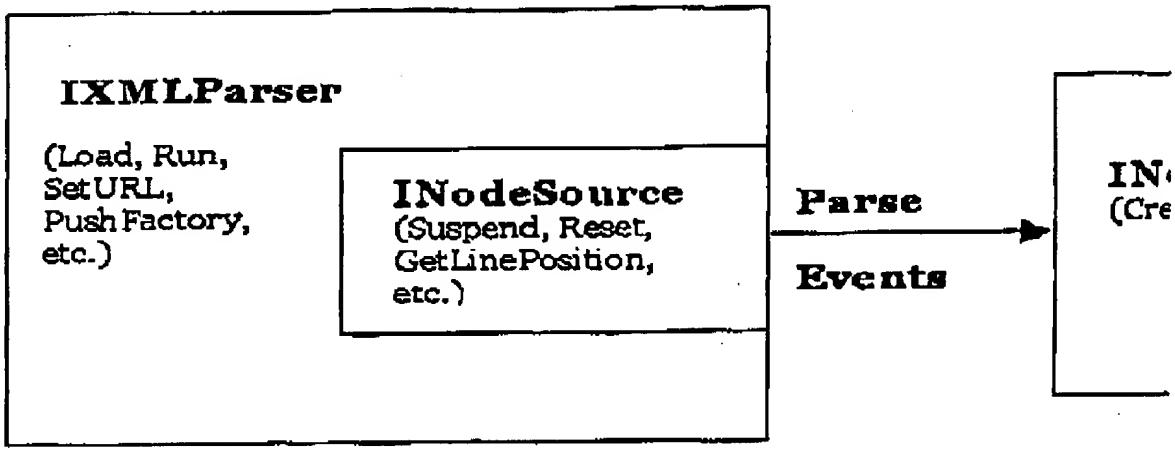
The Node Factory is not the only way for applications to deal with XML. The IDOMNode interface (see XMLDOM.htm) maintains an in-memory tree representation of an XML document, and provides methods for navigating, querying (see ../XQL Control Object Model.doc), and modifying the tree. This OM is generally simpler for most applications. In general, if your application does not need the XML processor to remember the contents of the XML document -- either because you have your own representation, or you are processing the document and discarding it, then it is more efficient to use the NodeFactory approach. Otherwise, the IDOMNode will be easier to use.

Information about XML in general is available from http://www.microsoft.com/standards/xml/ and information about our XML plans, IE5 XML, etc is available from http://xmlweb.

# Contents

# Parser

The Parser takes XML input in a variety of ways (i.e. via a stream, via a URL to a document, or via text pushed to it), parses the XML and sends parse events to a NodeFactory. The parser is divided into two interfaces: an INodeSource, which defines the parse events and other information (such as position information for parse errors) that are sent to a NodeFactory. The IXMLParser interface which inherits from INodeSource and adds methods to define the XML source (stream, URL, or pushed text), set and push NodeFactories, and deal with security and state reporting issues.




```
interface INodeSource: IUnknown
{
    HRESULT Suspend();

    HRESULT Reset();

    ULONG GetLineNumber();

    ULONG GetLinePosition();

    ULONG GetAbsolutePosition();

    HRESULT GetLineBuffer(
        [out] const WCHAR** ppwcBuf,
        [out] ULONG* pulLen);

    HRESULT GetLastError();

    ULONG GetErrorPosition();

    HRESULT GetErrorReason(
        [out] BSTR* pbstrReason);

    HRESULT SetFlags(
        [in] ULONG iFlags);

    ULONG GetFlags();
}
```

Push Model XML Parser / Node Factory

```
interface IXMLParser : INodeSource
{
    HRESULT SetURL(
        [in] const WCHAR* pszBaseUrl,
        [in] const WCHAR* pszRelativeUrl,
        [in] BOOL fAsync);

    HRESULT Load(
        [in] BOOL fFullyAvailable,
        [in] IMoniker *pimkName,
        [in] LPBC pibc,
        [in] DWORD grfMode);

    HRESULT SetInput(
        [in] IUnknown *pStm);

    HRESULT PushData(
        [in] const char* pData,
        [in] ULONG ulChars,
        [in] BOOL fLastBuffer);

    HRESULT SetRoot(
        [in] IUnknown* pUnkRoot);

    HRESULT PushFactory(
        [in] INodeFactory* pNodeFactory);

    HRESULT GetFactory(
        [out] INodeFactory** ppNodeFactory);

    HRESULT Run(
        [in] long lChars);

    HRESULT GetReadyState();

    HRESULT Abort(
        [in] BSTR bstrErrorInfo);

    HRESULT GetURL(
        [out] const WCHAR** ppwcBuf);

    // These are still needed to set the secure base URL.  Perhaps rename
    // to make this clearer.
    // This is used as a default if LoadDTD or LoadEntity or SetURL is ca
    // with a NULL base URL.
    HRESULT SetSecureURL(
        [in] const WCHAR* pszBaseUrl);

    HRESULT GetSecureURL(
        [out] const WCHAR** ppwcBuf);
};
```

The parser is multithread safe. In other words, it is safe to call these methods from any thread.

**SetURL**

This is one of four different methods for providing input to the parser. You pass a base url, and a relative url. For example, the base could be "http://www.microsoft.com/xml/test.htm" and the relative url could be "test.xml". The async flag controls whether an http request is handled synchronously or asynchronously. If it is async, you will get E_PENDING from Run and you will need to make sure you pump your message queue. (URLMon requires that you pump the

message queue).

**Load**

This method corresponds to the Load method in IPersistMoniker. The parser will call BindToStorage to get an IStream and load the XML associated with the given moniker. You can also call GetURL to get the URL representation of the given Moniker.

**SetInput**

Use this method if you already have an IStream containing XML. If the stream returns E_PENDING then the parser returns E_PENDING from Run(). The caller then must call Run again at a later time when more data is available.

**PushData**

The lowest level way to provide input is as a raw buffer of chars. The buffer is not NULL terminated. You call PushData with each buffer and set lastBuffer to TRUE when you push the last buffer. Run() will continue to return E_PENDING until you do this. It is also ok for a token to span multiple buffers. For example, the first buffer might end with "<!-- foo" in which case the parser will return E_PENDING, then if the next buffer starts with "bar--->" then the parser will complete the token and call the NodeFactory with the combined COMMENT with text "<!—foobar—>".

**SetRoot**

This sets the top-most node which is used as the pUnkParent in all top level CreateNode calls. The XML Object Model, for example, could pass in the IXMLDocument for this.

**PushFactory**

The parser delegates to a given NodeFactory for each element it finds in the XML source. See NodeFactory Interace for details. More than one node factory can also be pushed and they will be automatically popped by the parser based on the scope in which they were pushed. This could be used, for example, to provide different node factories to handle different namespaces within the document.

**Run**

Run parses the specified amount of XML (in characters), then it returns E_PENDING if it isn't finished or it returns S_OK when it reaches the end of the input. An amount of -1 means parse as much as it can get from the input stream. If you are using PushData, the Run method returns E_PENDING until you push the last buffer and then after that it will return S_OK. The number given to Run is just a hint -- if there are not the specified number of characters left in the input buffer then Run will still succeed.

If you used SetURL or Load to load a remote XML file, Run(-1) will return E_PENDING, but then so long as your application has a message loop, the download and parsing will occur in the background as data becomes available from URLMON. Then you can check GetReadyState

from time to time and at some point the parser will no longer return XMLPARSER_BUSY. If you want to monitor the progress you can use the Load method and register your own IBindStatusCallback and watch the OnProgress calls.

**GetReadyState**

GetReadyState returns one of the following values indicating the state of the parser:

- IDLE if the parser is in the Reset state,
- WAITING if the input stream returned E_PENDING ,
- BUSY if there is data available for parsing,
- ERROR if the parser found an error,
- STOPPED if the parser was aborted and
- SUSPENDED if it is currenly suspended.

**Abort, Suspend, Reset**

The parser can be stopped by the caller or by the NodeFactory at anytime by calling Abort. If the NodeFactory calls abort, it can also provide an IErrorInfo object that contains more information about why the node factory is aborting the parse. This will then be returned from GetErrorInfo.

The parser can also be suspended at any time and resumed by calling Run again. This helps clients tweak their performance by doing just-in-time parsing.

The Reset method puts the parser back into the initial state so you can load another XML file. Reset also resets the root object and the root NodeFactory and all other Node Factories that may be in the current parser context.

**GetLineNumber, GetLinePosition, GetLineBuffer, GetAbsolutePosition, GetLastError, GetErrorInfo, GetErrorPosition**

The current line position information can be queried at any time. The INodeFactory is given a pointer to the IXMLParser so that it can use these functions also. GetAbsolutePosition returns the absolute offset from the beginning of the input stream (or buffer) (after UTF-8 decoding has been applied). GetLastError returns the last HRESULT that was returned from other methods and GetErrorInfo returns a BSTR which contains more descriptive information about what went wrong. GetPosition returns the offset of the error in characters from the start of GetLineBuffer's line buffer.

**SetFlags,GetFlags**

The following flags can be OR'd together to control how the parser works. By default none of these flags are set.

**XMLFLAG_NONAMESPACES**
Whether the namespace declarations and tag name syntax are recognized.
**XMLFLAG_NOWHITESPACE**
Whether to return WHITESPACE nodes.

**XMLFLAG_CASEINSENSITIVE**

This makes the parser case insensitive - but the text you get is still whatever was in the original file. It does NOT fold to upper case since this is inefficient and is not always needed.

**XMLFLAG_IE4COMPATIBILITY**

Turn on all IE4 compatbility flags, which is an OR of all of the above (XMLFLAG_CASEINSENSITIVE, XMLFLAG_NONAMESPACES, XMLFLAG_NOWHITESPACE) plus:

- allows PCDATA to contain unescaped ampersand characters
- allow short end tags </>
- normalize whitespace in attribute values and PCDATA
- allowing SGML comment syntax ('--') inside a comment
- allow duplicate attributes
- allow comments before the xml declaration, and so forth

Note that enabling the IE4 Compatability makes the parser run 11% slower.

**XMLFLAG_PROCESSNAMESPACES**

Whether to provide namespace processing. See Standard Namespace Processing.

**The stuff below this line isn't being implemented for IE5. However we're keeping the design here for the release after IE5.**

**XMLFLAG_PROCESSSCHEMAS**

Whether to provide schema processing. This flag causes the parser to automatically plug in the Standard Schema Proecessing

**XMLFLAG_VALIDATE**

Tells the schema processor whether to report validation errors against any schema present in the document instance. Note that it is perfectly sensible to request validation without Schema information, Schema information without validatin, or both validation and schema information.

**XMLFLAG_EXPANDENTITIES**

If this flag is set, the XMLParser expands all entity references for the customer node factory. If this flag is off, the behavior is as it was for Beta 1 -- the customer NodeFactory receives CreateNode calls on ENTITYREF objects (i.e. for each "&foo;"). Setting this flag makes it easier to write the common case where the customer NodeFactory application need not be aware of what content came from entities as opposed to in-line text. Note when Entity expansion is off, the XMLParser will call the customer node factory with a stream of events representing the XML content that comes from entity declarations. These events will be delivered immediately after all schemas are processed, but before the root node of the XML Document (note this is **not** when the entity declaration is encountered in the DTD or Schema -- for obscure XML reasons the parse nature of the entity could be changed by later schema information, so all entities must be processed after all other schema information). This will give the customer node factory that is doing it's own entity processing a chance to construct and save entity information in it's own format, possibly to refer to when ENTITYREF events are received later in the document.

**GetURL**

In the case where SetURL() or Load() was called, this method returns the URL that the parser

is loading.

## Encoding Support

For all the different methods of input, the same enoding support is provoided. To provide unicode data the input stream or buffer must start with a Unicode Byte Order mark 0xFFFE or 0xFEFF - depending on whether the unicode is little or big endian. If it doesn't start with a byte order mark it is assumed to be UTF-8. To specify something other than UTF-8 use an xml declaration with an encoding attribute at the beggining of the input. For example, the following specifies a common encoding: for Japanese on Microsoft platforms:

```
<?xml version="1.0" encoding="shift_jis"?>
```

The XML Parser recognizes UTF-8 and UCS-2 directly. For all other character sets, MLang (www.microsoft.com/msdn/sdk/inetsdk/help/itt/mlang/overview.htm) is used to perform conversions. A complete list of character sets supported by MLang is generally found at http://ie/Specs/Intl/charsets.htm.

## Example Parser Usage

So an example usage if you have an IStream and a NodeFactory already is as follows:

```
HRESULT ParseXML(IStream* input, INodeFactory* f)
{
    IXMLParser* xp;
    HRESULT hr;
    checkhr(CoCreateInstance(CLSID_XMLParser, NULL, CLSCTX_INPROC_SERVER,
                             IID_IXMLParser, (void**)&xp));

    checkhr(xp->SetInput(input));
    checkhr(xp->PushFactory(f));
    hr = xp->Run(-1);
    xp->Release();
    return hr;
}

interface INodeFactory : IUnknown
{
        HRESULT NotifyStart(
                [in] INodeSource *pNodeSource,
                [in] NodeFactoryEvent eEvent);

        HRESULT NotifyEnd(
                [in] NodeFactoryEvent eEvent);

        HRESULT EndChildren(
            [in] INodeSource *pNodeSource,
            [in] BOOL fEmptyNode,
            [in] DWORD dwType,
            [in] const WCHAR* pwcText,
            [in] ULONG ulLen,
            [in] IUnknown* pUnkNode);

        HRESULT Error(
            [in] INodeSource *pNodeSource,
            [in] HRESULT hrErrorCode);

        HRESULT CreateNode(
```

```
                [in] INodeSource *pNodeSource,
                [in] IUnknown* pUnkParent,
                [in] IUnknown* pUnkOuter,
                [in] DWORD dwType,
                [in,out] DWORD* dwSubType,
                [in] const WCHAR* pwszText,      // tag name
                [in] ULONG ulText,               // tag name length
                [in] const WCHAR* pwNsPrefix,    // namespace prefix
                [in] ULONG ulNsPrefix,           // prefix length
                [in] NameSpace *pNS,             // namespace object
                [in] IUnknown *pSchema,          // Schema object (future)
                [out] IUnknown** ppUnkChild);
);
```

The NodeFactory is responsible for creating nodes based on the information provided by the XML Parser. This is essentially a callback interface so that custom node factories can be provided that build different kinds of object hierarchies. Notice that the nodes returned are any IUnknown object - which means this is not tied specifically to the XML Object Model in any way. The XML Parser assumes nothing about the returned IUnknown objects which means the NodeFactory can also return NULL.

### NotifyStart, NotifyEnd

Convenience methods where the NodeFactory can do some initialization and cleanup. This is particularly useful if the same NodeFactory is being used to load multiple documents. Takes a NodeFactoryEvent parameter, which is one of:

```
typedef enum
{
        NFE_DOCUMENT = 1,   // document started or ended
        NFE_DTD.        // external DTD started or ended.
        NFS_INTERNALSUBSET, // internal subset DTD started or ended
        NFE_EXTERNALENTITY,
} NodeFactoryEvent;
```

### EndAttributes

This method is called when all the attributes for a given element are complete. In other words the greater-than character (>) has been reached. This method is not called for terminal nodes but may be called for empty elements (e.g. <foo id="123"/>).

### EndChildren

This method is called when all the subelements of the given element are complete. In other words the matching end tag </FOO> has been reached. This is also called if the tag is an empty tag <FOO ... /> in which case the fEmpty argument is set to TRUE in case the NodeFactory needs to distinguish between this case and <FOO></FOO>. This method is not called for terminal nodes.

### Error

This method is called when the parser runs into an error in the XML document. The parser will stop at this point and return the HRESULT error code to the caller. The NodeFactory can call back to the parser to get more information about the error.

**CreateNode**

CreateNode is the main method that gets called during parsing for every element, attribute and attribute value. This is geared towards the new w3c Document Object Model (DOM) where Attributes are themselves Nodes, and attribute values can also consist of multiple child nodes.

**IXMLParser* pParser**
The parser is passed into each node factory call so that the node factory can call back and get important information - like the current line number, or to stop the parser.

**IUnknown* pUnkParent**
This is the parent object of the node being created. This parent pointer was returned from a prior CreateNode call, or is the root object provided using the parser SetRoot method.

**IUnknown* pUnkOuter**
If the created node is part of and aggreagte the pointer pUnkOuter is the controlling IUnknown, otherwise pUnkOuter is NULL. In case of aggregation the returned ppUnkChild is the IUnknown of the created node and QI on ppUnkChild will yield the desired type - for example, IDOMNode.

**DWORD dwType**
The node type. See table of node types below in CreateNode Reference.

**DWORD dwSubType**
The node sub type. See table of node types below in CreateNode Reference.

**const WCHAR* pwcText**
This is either a tag name, or a PCDATA text value. The lifetime of this pointer is the lifetime of the IXMLParser that is the driving application. If the text is not a tag name, then the lifetime is even shorter -- only the duration of this CreateNode call. Note that Element/Attribute/PI tag names or certain attribute values (of type ID, NMTOKEN, ENTITY, or NOTATION), may have namespace prefixes. For this parameter, the prefixes will be stripped off and passed in the pwcNsPrefix intead. E.g. if a tag name is "foo:bar", then this argument will get "bar", and pwcNsPrefix will get "foo".

**ULONG ulLen**
This the length of the text argument.

**const WCHAR pwcNsPrefix**
This is the namespace part of the tag name or attribute value (see explanation under pwctext). This string is also NUL terminated, and "atomized".

**ULONG ulLen**
This the length of the pwcNsPrefix argument.

**IUnknown** ppUnkChild**
The NodeFactory may or may not create a node object representing the above information and return it here. If a non-NULL pointer is returned then this may be used as the parent in subsequent calls to CreateNode. In other words, the XML Parser maintains the parse context for you and passes in the appropriate parent pointer based on what it finds in the XML. Notice also

that this is an IUnknown -- so the NodeFactory need not be building the XML Object Model.

**NameSpace * pNS**

Points to an object of type namespace (see Standard Namespace Processing below), used to communicate the full namespace information about the current name. Unless XMLFLAG_PROCESSNAMESPACES is set, this parameter is always NULL.

**IUnknown *pSchema**

For IE5 always NULL. In the future:
For element and attribute types only, points to an IDOMNode object. which is an XML-Data style schema node representing the schema information for the node being built. See Standard Schema Processing below. Unless XMLFLAG_PROCESSSCHEMAS is set, this parameter is always NULL.

**HRESULT**

If anything other than S_OK is returned, then the parser stops and returns the same error from the parser Run() method. So you could return E_PENDING if your node factory itself is loading some other XML file and is still waiting for data. In fact, this is exactly what a DTDNodeFactory would do.

So for example, given the following XML fragment:

```
<item id="FOO" ms:price="20" title="BAR &foo;">
        <value>The quick brown fox</value>
</item>
```

The following sequence of calls will be made on the node factory (indented for readability only - does not imply recursion):

This probably needs updating. -cfranks

```
CreateNode(parser, root, ELEMENT, "item", 4, 0, &item);
  CreateNode(parser, item, ATTRIBUTE, "id", 2, 0, &id);
    CreateNode(parser, id, PCDATA, "FOO", 3, &value);
  EndChildren(parser, id);
  CreateNode(parser, item, ATTRIBUTE, "ms:price", 8, 2, &price);
  CreateNode(parser, price, PCDATA, "20", 2, &value);
  EndChildren(parser, price);
  CreateNode(parser, item, ATTRIBUTE, "title", 5, 0, &title);
    CreateNode(parser, title, PCDATA, "BAR ", 3, &value);
    CreateNode(parser, title, ENTITYREF, "foo", 3, &value);
  EndChildren(parser, title);
EndAttributes(parser, item);
  CreateNode(parser, item, WHITESPACE, "0xd 0xa 0x9", 3, &value);
  CreateNode(parser, item, ELEMENT, "value", 5, 0, &value);
    CreateNode(parser, value, TEXT, "The quick brown fox", 19, &text);
    EndAttributes(parser, value);
  CreateNode(parser, item, WHITESPACE, "0xd 0xa", 2, &value);
EndChildren(parser, value);
  EndAttributes(parser, item);
  EndChildren(parser, item);
EndChildren(parser, root);
```

So notice that attributes can also have children, just like elements.

## Example NodeFactory

I'm sure this needs updating. -cfranks

A NodeFactory for the CDF Viewer that scans for the LOGO ICON HREF would be implemented with the following pseudo-code:

```
class IFindIconNodeFactory : public INodeFactory
{
    ULONG   level;
    bool    inLogoTag;
    BSTR    attribute;
    BSTR    href;
    bool    found;
    int     state;

    IFindIconNodeFactory()
    {
        level = 0; inLogoTag = false;
        href = NULL; found = false; attribute = NULL;
    }

    HRESULT createNode(IXMLParser* p, IUnknown* parent,
                DWORD dwType, DWORD dwSubType,
                const WCHAR* text, ULONG len, ULONG nslen,
                IUnknown**pChild, NameSpace *pNS)

    {
        if (! terminal)
        {
            level++;
            if (level == 2 && dwType == XML_ELEMENT && strcmpi(text, "ELEMENT)
            {
                inLogoTag = true;
            }
            else if (inLogoTag && dwType == XML_ATTRIBUTE)
            {
                ::SysFreeString(attribute);
                attribute = ::SysAllocStringLen(text, len);
            }
        }
        else
        {
            if (inLogoTag && dwType == PCDATA)
            {
                if (strcmpi(attribute, "STYLE")==0 && strcmpi(text, "ELEMENT)=
                {
                    // we found what we're looking for but we don't
                    // terminate until all attributes are complete so
                    // that we are sure to get the HREF attribute no
                    // matter what order the attributes were specified.
                    found = true;
                }
                else if (strcmpi(attribute, "HREF")==0)
                {
                    ::SysFreeString(href);
                    href = ::SysAllocStringLen(text, len);
                }
            }
            *pChild = NULL; // No tree creation !!
```

```
        return S_OK;
    }

    HRESULT EndChildren(IXMLParser* p, IUnknown* node)
    {
        level--;
    }

    HRESULT EndAttributes(IXMLParser* p, IUnknown* node)
    {
        if (inLogoTag && found)
        {
            p->Abort(NULL); // We're done !!
        }
        inLogoTag = false;
    }

    // Other methods left out of example since they are not needed in this cas

}
```

What this does is scan through the following XML to extract the highlighted information:

```
<?XML VERSION="1.0"?>
<CHANNEL HREF="http://www.msnbc.com">
<SELF HREF="http://www.microsoft.com/ie/ie40/msnbc/msnbc.cdf" />
<TITLE>The MSNBC Channel</TITLE>
<LOGO HREF="http://www.microsoft.com/ie/ie40/msnbc/msnbc.gif" STYLE="IMAGE"/>
<LOGO HREF="http://www.microsoft.com/ie/ie40/msnbc/msnbc.ico" STYLE="ICON"/>
...
```

Then it aborts the parse -- so this will usually complete within the first buffer of input clearly a lot more optimal than loading the entire document.

## Standard Namespace Processing

If XMLFLAG_PROCESSNAMESPACES is set, then the parser does namespace processing prior to the user's node factory receiving events. All that this means is that the processor will recognize the <?xml:namespace ... > PI and remember it's contents in a NameSpace object. Note that all the strings in the namespace object are NUL terminated, and "atomized" -- meaning that equal string values will have equal pointer values (i.e. there is only one copy of a string). **The lifetime of a NameSpace object passed down in a CreateNode call is the lifetime of the CreateNode call.** Customer Node Factories should not remember the CreateNode object. However, the strings pointed to by the member variables of the NameSpace object have the same lifetime as the IXMLParser.

```
class NameSpace
{
    const WCHAR * szNS;     // the ns part of a namespace PI
    const WCHAR * szPrefix; // the prefix part of a namespace PI
    const WCHAR * szSrc;    // the source part of a namespace PI
}
```

## Standard Schema Processing

NOT BEING IMPLEMENTED FOR IE5.

If XMLFLAG_PROCESSSCHEMAS is set. then the parser does schema processing prior to the user's node factory receiving events. The parser will then handle all schema processing -- XML-Data schemas referenced by namespace PIs, and DTDs referenced by namespace PIs or DOCTYPE (both internal and external subsets). Schema information is passed to the customer node factory in the pSchema parameter, which points to an IDOMNode object that is the XML-Data style representation of the schema for that node (see XMLDataSubset.htm, and SchemaNavigation.htm). DTD schema are converted to XML-Data schema representation internally. The following events will be accompanied by Schema information (by dwType):

| dwType | pReserved parameter |
|---|---|
| XML_ELEMENT | IDOMNode * pointing to the <ElementType ...> declaration for the element. |
| XML_ENTITYREF | (Only if XMLFLAG_EXPANDENTITIES is 0). IDOMNode * pointing to <Entity ...> declaration for the entity. Note for external parsed entities the contents of the external parsed entity will be available by following the definition parameter of the entity to the loaded document (if available) containing the external parsed entity. |
| XML_PI | (Subtype == XML_NAMESPACEDECL, i.e. namespace PI only). IDOMNode * pointing to the root of the external schema (if found). I.e. <Schema ...>. |
| XML_DOCTYPE | IDOMNode * pointing to the root of an XML-Data style schema formed by merging the external and internal subsets, and converting both to an XML-Data style schema. (Note that parameter entities are of course expanded before this happens -- their original definition cannot be recovered). |

With this flag on, customer node factories will receive no events concerning the internals of schema information. I.e. if there is an internal DTD subset in the document, it will be internally processed by the Schema Node Factory. and the customer node factory will receive no events for these. This decision is made because we believe that customer node factories will by and large have such little interest in or knowledge of DTD syntax that it is best not to even burden them with filtering it out.

# CreateNode Reference

This is the complete set of node types and when they occur. There are two main types of nodes, those that can have children (containers) and those that cannot (terminal nodes).

The second column in the tables below, labelled "Which node factories will output this tag?" says which factories will call CreateNode on that tag type. If a node factory doesn't show up in a particular column, that means it filters that node type. This really only applies to the schema node factory, which will filter out all namespace PI's and Schema information.

(The dwSubTypes in bold aren't in Chris Lovett's IDL file yet -- I think we need these).

## Containers

| dwType | dwSubType | Which node factories will output this tag? | When occurs |
|---|---|---|---|
| XML_ELEMENT | 0 | standard, namespace, schema | At the start of a new element <foo... |
| XML_ATTRIBUTE | 0 | standard, namespace, schema | At the start of a new attribute FOO="..." |
| | XML_SYSTEM | standard, namespace | SYSTEM literal (applicable in <!ENTITY> or <!DOCTYPE> only) |
| | XML_PUBLIC | standard, namespace | PUBLIC literal (applicable in <!ENTITY> or <! DOCTYPE> only) |
| | XML_VERSION | standard, namespace | XML *version* string (applicable in <?xml> PI only) |
| | XML_ENCODING | standard, namespace | XML *encoding* string (applicable in <?xml> PI only) |
| | XML_STANDALONE | standard, namespace | XML *standalone* string (applicable in <?xml> PI only) |
| | XML_NS | standard, namespace | xml:namespace *ns* part (applicable in <? xml:namespace PI only) |
| | XML_SRC | standard, namespace | xml:namespace *src* part (applicable in <? xml:namespace PI only) |
| | XML_PREFIX | standard, namespace | xml:namespace *prefix* part (applicable in <? xml:namespace PI only) |
| | XML_XMLSPACE | standard, namespace, schema | The special global xml:xmlspace attribute. |

| | | | |
|---|---|---|---|
| | XML_XMLLANG | standard, namespace, schema | The special global xml:xmllang attribute. |
| XML_PI | 0 | standard, namespace, schema | The start of a <?foo with text equal to PI name. |
| | XML_XMLDECL | standard, namespace | This the special <?xml ... PI. In this case the parser parses the version, encoding and standalone attributes and passes these as special node types. See XML_VERSION, XML_ENCODING and XML_STANDALONE below. |
| | XML_NAMESPACEDECL | standard, namespace | This the special <? xml:namespace... PI. In this case the parser parses the name, href and as attributes and passes these as special node types. See XML_NSNAME, XML_HREF and XML_AS below. |
| XML_DOCTYPE | 0 | standard, namespace | This is the special <! DOCTYPE declaration. The text is the doctype name and this will be followed by an optional XML_PUBLIC or XML_SYSTEM node and an optional XML_DTDSUBSET. If there is a 'PUBLIC' keyword you will get XML_PUBLIC with the text of the public id followed by XML_SYSTEM with the text of the system literal. |

**Terminal Nodes**

| dwType | dwSubType | Which node factories will output this tag? | When occurs |
|---|---|---|---|
| XML_TEXT | XML_PCDATA | standard, namespace, & schema | The text inside a node or an attribute. |
| | XML_CDATA | standard, namespace, & schema | The text inside <! [CDATA[...]]> |
| | XML_WHITESPACE | standard, namespace, & schema | White space between elements |
| | XML_ID | standard, namespace, & schema | an attribute value of type ID |
| | XML_IDREF | standard, namespace, & schema | an attribute value of type IDREF |
| | XML_ENTITY | standard, namespace, & schema | an attribute value of type ENTITY |
| | XML_NMTOKEN | standard, namespace, & schema | an attribute value of type NMTOKEN |
| | XML_NOTATION | standard, namespace, & schema | an attribute value of type NOTATION |
| | XML_NAME | standard, namespace | general name token for typed attribute values or DTD declarations |
| | XML_STRING | standard, | general quoted string literal in DTD |

| | | | |
|---|---|---|---|
| | | namespace | declarations. |
| XML_COMMENT | | standard, namespace, & schema | The text inside '<!--' and '-->' |
| XML_ENTITYREF | XML_NAMEDENTITYREF | standard, namespace, & schema (with dontexpandentities flag on) | A named entity node, &foo; |
| | XML_GENERALENTITYREF | standard, namespace, & schema (with dontexpandentities flag on) | A numeric entity, &#23; |
| | XML_NUMENTITYREF | standard, namespace & schema (with dontexpandentities flag on | A numeric entity, &#23; |
| | XML_HEXENTITYREF | standard, namespace & schema (with dontexpandentities flag on | A hexidicimal entity, ೷ |

## Errors

```
typedef enum
{
    XML_E_PARSEERRORBASE = 0x00000000, // NEED AN OFFICIAL RANGE!

    XML_E_ENDOFINPUT            = XML_E_PARSEERRORBASE,
    XML_E_UNCLOSEDPI,           // 1
    XML_E_MISSINGEQUALS,        // 2
    XML_E_UNCLOSEDSTARTTAG,     // 3
    XML_E_UNCLOSEDENDTAG,       // 4
```

```
XML_E_UNCLOSEDSTRING,          // 5
XML_E_MISSINGQUOTE,            // 6
XML_E_COMMENTSYNTAX,           // 7
XML_E_UNCLOSEDCOMMENT,         // 8
XML_E_BADSTARTNAMECHAR,        // 9
XML_E_BADNAMECHAR,             // A
XML_E_UNCLOSEDDECL,            // B
XML_E_BADCHARINSTRING,         // C
XML_E_XMLDECLSYNTAX,           // D
XML_E_BADCHARDATA,             // E
XML_E_UNCLOSEDMARKUPDECL,      // F
XML_E_UNCLOSEDCDATA,           // 10
XML_E_MISSINGWHITESPACE,       // 11
XML_E_BADDECLNAME,             // 12
XML_E_BADEXTERNALID,           // 13
XML_E_EXPECTINGTAGEND,         // 14
XML_E_BADCHARINDTD,            // 15
XML_E_BADELEMENTINDTD,         // 16
XML_E_BADCHARINDECL,                    // 17
XML_E_MISSINGSEMICOLON,        // 18
XML_E_BADCHARINENTREF,         // 19
XML_E_UNBALANCEDPAREN,         // 1A
XML_E_EXPECTINGOPENBRACKET,    // 1B
XML_E_BADENDCONDSECT,          // 1C
XML_E_RESERVEDNAMESPACE,       // 1D
XML_E_INTERNALERROR,           // 1E
XML_E_EXPECTING_VERSION,        // 1F
XML_E_EXPECTING_ENCODING,       // 20
XML_E_UNEXPECTED_ATTRIBUTE_IN_NAMESPACEDECL,        // 21
XML_E_EXPECTING_NAME,          // 22
XML_E_NAMESPACEDECLSYNTAX,     // 23
XML_E_UNEXPECTED_WHITESPACE,   // 24
XML_E_UNEXPECTED_ATTRIBUTE,    // 25
XML_E_SUSPENDED,               // 26
XML_E_STOPPED,                 // 27
XML_E_UNEXPECTEDENDTAG,        // 28
XML_E_ENDTAGMISMATCH,          // 29
XML_E_UNCLOSEDTAG,                 // 2A
XML_E_DUPLICATEATTRIBUTE,      // 2B
XML_E_MULTIPLEROOTS,           // 2C
XML_E_INVALIDATROOTLEVEL,      // 2D
XML_E_BADXMLDECL,              // 2E
XML_E_INVALIDENCODING,         // 2F
XML_E_INVALIDSWITCH,           // 30
XML_E_MISSINGROOT,             // 31
XML_E_INCOMPLETE_ENCODING,     // 32
XML_E_EXPECTING_NDATA,         // 33
XML_E_INVALID_MODEL,            // 34
XML_E_BADCHARINMIXEDMODEL,     // 35
XML_E_MISSING_STAR,            // 36
XML_E_BADCHARINMODEL,          // 37
XML_E_MISSING_PAREN,            // 38
XML_E_INVALID_TYPE,            // 39
XML_E_INVALIDXMLSPACE,         // 3A
XML_E_MULTI_ATTR_VALUE,        // 3B
XML_E_INVALID_PRESENCE,         // 3C
XML_E_BADCHARINENUMERATION,    // 3D
XML_E_UNEXPECTEDEOF,           // 3E
XML_E_BADPEREFINSUBSET,        // 3F
XML_E_BADXMLCASE,              // 40
XML_E_CONDSECTINSUBSET,        // 41
XML_E_CDATAINVALID,            // 42
XML_E_INVALID_STANDALONE,      // 43
XML_E_PE_NESTING,              // 44
```

```
    XML_E_UNEXPECTED_STANDALONE,     // 45
    XML_E_DOCTYPE_IN_DTD,          // 46
    XML_E_INVALID_CDATACLOSINGTAG, // 47
    XML_E_PIDECLSYNTAX,            // 48
    XML_E_EXPECTINGCLOSEQUOTE,   // 4C
    XML_E_DTDELEMENT_OUTSIDE_DTD, // 4A
    XML_E_DUPLICATEDOCTYPE, // 4B
    XML_E_MISSING_ENTITY,          // 4C
    XML_E_ENTITYREF_INNAME,        // 4D
    XML_E_LASTERROR,
} XMLErrorCode;

// Possible ready states
typedef enum
{
    XMLPARSER_IDLE,
    XMLPARSER_WAITING,
    XMLPARSER_BUSY,
    XMLPARSER_ERROR,
    XMLPARSER_STOPPED,
    XMLPARSER_SUSPENDED
} XMLReadyState;

// Some parser flags which can be OR'd together.
// By default all flags are clear.
typedef enum
{
    XMLFLAG_CASEINSENSITIVE   = 1,
    XMLFLAG_NONAMESPACES      = 2,
    XMLFLAG_NOWHITESPACE      = 4,
    XMLFLAG_IE4COMPATIBILITY  = 8,
    XMLFLAG_VALIDATION        = 16, // whether to load and process DTD's.
    XMLFLAG_PROCESSNAMESPACES = 32,
    XMLFLAG_PROCESSSCHEMAS    = 64,
    XMLFLAG_EXPANDENTITIES    = 128,
} XMLParserFlags;
```

# Current Status

As of early June 1998, we're starting Beta 2 work.

# Open Issues

1.

# Change History

| | |
|---|---|
| 1998.06.18 | Rev 1.21: Update for all the Beta 2 schedule cuts. No support for XMLFLAGS_PROCESSSCHEMA, XMLFLAGS_VALIDATE, or XMLFLAGS_EXPANDENTITIES. Remove stuff about atomizing tag names. |
| 1998.06.1 | Rev 1.15: Stop identifying the Namespace and Schema Nodefactories as distinct node factories. They are now just standard parser features. Eliminate pReserved parameter in favor of pNS and pSchema. Fix up flag definitions. |
| 1998.06.09 | Extensive editing in response to CLovett feedback. Nul terminated and atomized CreateNode strings for Java compatibility. |
| | PM (CFranks) finally takes over this document. First round of documenting |

| | |
|---|---|
| 1998.06.06 | Beta 2 changes: standard namespace and schema node factories, dwType and dwSubType on CreateNode, automatic entity expansion feature of schema node factory. |
| 3/19/98 | Fixed buffer overrun bug in PushData usage, and added xml:space and xml:lang implementation. So for these attributes you now get the XML_XMLSPACE and XML_XMLLANG node type instead of XML_ATTRIBUTE. . |
| 3/15/98 | Added loadDTD, and added new parser flags. . |
| 3/9/98 | Added StartDTDSubset and EndDTDSubset and added implementation for GetErrorInfo. . |
| 3/4/98 | Added StartDTD and EndDTD and added type result field to CreateNode. Also added GetURL. |
| 3/1/98 | Added Load method for IPersistMoniker clients, and added flags to control how parser works and added unique attribute checking to parser implementation. |
| 2/28/98 | Added SetURL(base,url) for secure XML download support. |
| 2/27/98 | Fixed memory leaks and added new void* reserved pointer to CreateNode. |
| 2/27/98 | Changed Reset to also reset NodeFactory and Root node. |
| 2/26/98 | Added new args to EndChildren and EndAttributes. |
| 2/25/98 | Implemented encoding support. Provided first implementation for testing with. Added GetAbsolutePosition. Changed Get:LastError to return the last HRESULT and added GetErrorInfo. Added more clarifications based on feedback so far. |
| 2/19/98 | Changed error handling to use IErrorInfo and changed GetLastError to just return HRESULT and added GetErrorInfo to return the IErrorInfo. Also changed Run to take a long number of characters instead of killobytes. Added Suspend/Resume and removed SetURL. Changed name of interface from IParser to IXMLParser. Removed Resume since this is redundant with Run. Added Reset. |